JOURNAL OF SOFTWARE MAINTENANCE: RESEARCH AND PRACTICE
J. Softw. Maint: Res. Pract. **11**, 167–182 (1999)

Practice

Dynamic Analysis Methods for the Year 2000 Problem

NORMAN WILDE[1]* RANDY JUSTICE[2], KRISTIN BLACKWELL[3] and W. ERIC WONG[4]
[1]*Department of Computer Science, University of West Florida, Pensacola FL 32514–5732, U.S.A.*
[2]*DynTel Corp., 6490 Saufley Field Rd., Bldg. 2434, Pensacola FL 32509–5200, U.S.A.*
[3]*Motorola Inc., Cellular Systems Group, 1501 West Shure Drive, Arlington Heights IL 60004–1469, U.S.A.*
[4]*Bell Communications Research, 445 South St., Morristown NJ 07960–6438, U.S.A.*

SUMMARY

Programmers working on the year 2000 problem need to locate and understand date sensitive code, that is, code whose execution depends on date inputs. This paper presents several dynamic analysis methods for addressing this problem. Date sensitive code can be located by running many test cases that are identical except for a change in one date value. An advanced test coverage tool can recover each test's execution count vector giving the number of times that each basic block was executed. Comparison of the vectors reveals the blocks whose execution is affected by the input date values. We present several methods of analysing the execution count data to locate date algorithms. One method identifies subdomains, that is, ranges of dates that are processed similarly. Another method involves graphical or Fourier analysis to identify common programming patterns such as leap-year computations. The last method exploits the 28-year regularity in dates to look for anomalies in processing. The methods are illustrated using Bellcore's *ATAC* testing tool on five C programs that use some of the date encodings that year 2000 maintainers may need to locate and understand.

KEY WORDS: dynamic analysis; Y2K; software testing tools; white-box testing; pattern analysis; reverse engineering

1. INTRODUCTION

The so-called 'year 2000' problem has many facets. It is a testing problem, as legacy systems must be checked for their behaviour as the year changes from '99' to '00'. It is a certification problem, as every software-using organization must ensure that all acquired software is resilient to this same change; and it is a management problem, as many interrelated programs must be updated and phased into use before the crucial day.

But, at some point the year 2000 problem becomes a *programmer's* problem. Someone has to examine each module, check it for correctness under year 2000 conditions, and code any needed modifications. A big part of this programmer's problem is identifying *date sensitive code*, that is,

*Correspondence to: Dr. N. Wilde, Department of Computer Science, University of West Florida, 11000 University Parkway, Pensacola FL 32514–5732, U.S.A. Email: nwilde@uwf.edu
Contract/grant sponsor: NSF; Contract/grant number: EEC-9418762

CCC 1040–550X/99/030167–16$17.50

Received 17 July 1998
Revised 19 March 1999

those algorithms that process date information and that thus may contain faults that will become significant as the year 2000 approaches.

Dates cause the most subtle year 2000 problems when they affect control. Consider the following anecdote discussed informally in 1996 in the presence of one of the authors at a software industry conference.

> A certain insurance company noticed in early 1995 a marked decline in income from one of its policy lines. Investigation showed that policies were mysteriously being deleted from the company's files. Further investigation revealed that the problem was a faulty algorithm that implemented the business rule 'Delete any policy that has had no transactions in the last 5 years.' The algorithm was roughly as follows:
>
> ```
> Read policy's last transaction date
> If ( ( last transaction date + 5 years) < current date )
>      Delete policy
> ```
>
> Since the dates were encoded with a 2 digit year, adding 5 to 95 produced 00, which was interpreted as less than the current year of 1995. Any policy was thus deleted shortly after its first transaction in the year 1995.

The algorithm in this case is sensitive to the transaction date data item; the value of this item determines what code is executed. It also happens to be incorrect for certain values of the transaction date, that is, values with the year 1995 or later.

This paper will propose several ways of locating and understanding date sensitive code using *dynamic analysis*, that is, methods that involve running the code for different test cases. The methods that will be described need a good test coverage monitor and a good test harness; beyond that they only require a modest amount of 'glue' code to process the test coverage information through standard display and data analysis tools.

There is, of course, a huge and increasing volume of published material on the year 2000 problem, ranging from hype to serious analysis. There are papers that:

- discuss the scope of the problem and progress towards a solution (Thomas, 1998; Rubin, 1999),
- survey tools (Sharon, 1997; Zvegintzov, 1997),
- discuss process and management issues (Bohner, 1996; Kappelman, *et al.*, 1998; Gowan, Jesse and Mathieu, 1999),
- suggest how to prepare test data (Holmes, 1997),
- enumerate possible strategies for fixes (Martin, 1997; Lefkon, 1997),
- describe specific tools (Hart and Pizzarello, 1996; Newcomb and Scott, 1997), or
- provide case studies (Marcoccia, 1998).

This paper focuses on techniques of *dynamic analysis* to help the year 2000 programmer and builds on the *software reconnaissance* method of locating features within code (Wilde and Scully, 1995). Dynamic analysis of code involves running it, as opposed to *static analysis* of the source code without actual execution. (Many of the analysis tools suggested for the year 2000 problem are static in nature.) While all software testing is, of course, dynamic, the analysis is usually limited to

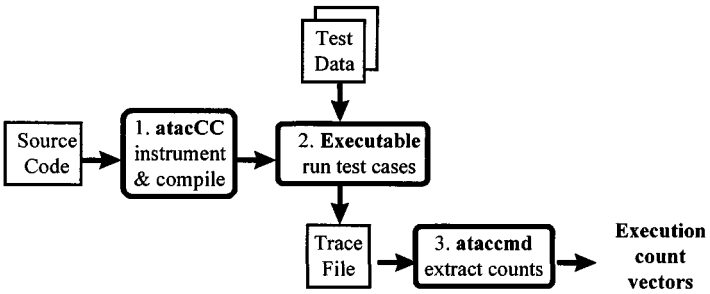

Figure 1. Using ATAC for Dynamic Analysis

checking the output for correctness. However, using a good *test coverage monitor*, it is possible to extract much more interesting information about a program from a set of well thought out test cases.

2. METHODOLOGY

Test coverage monitors are tools that have been used for some years to check the completeness of software testing. Testers want to know, for example, if a test set exercises all basic blocks in a program. (A *basic block* is a sequence of statements that has no transfers of control, so that if one statement is executed, all must be executed.) The test coverage monitor instruments the source code to insert counters at each basic block. The counters record the number of times the block was executed and are written to a *trace file* upon completion of each test.

The trace file thus holds a wealth of information about the program's execution that can be used for dynamic analysis. In this study we used a few of the facilities of Bellcore's *ATAC* testing tool (Horgan and London, 1992; Horgan, London and Lyu, 1994) following roughly the steps shown in Figure 1:

1. The programmer uses `atacCC` to instrument and compile the program. `AtacCC` inserts the counters and acts as a wrapper to execute the site's usual C compiler.
2. The programmer runs a series of test cases involving different dates. The counters for each run are saved to the trace file.
3. The programmer then runs scripts which call *ATAC*'s `ataccmd` command line interface to read the trace file and extract the count data.

The result is an *execution count vector* for each test, giving the number of times each basic block was executed. Note that while only basic blocks were used in this study, *ATAC* has many other features, for example reporting code coverage for decisions, c-uses, p-uses and all-uses. *ATAC* is part of χSuds^{TM}, a software understanding system developed at Bellcore. More information about χSuds^{TM} can be found at the URL: http://xsuds.bellcore.com

In the remainder of this paper we will describe several different ways of analysing the execution count vectors to provide year 2000 programmers with insight for program comprehension. The only other work we have encountered that uses dynamic analysis in this way is by Reps *et al.* (1997). In an approach similar to the white-box one in this paper, they instrument code to see if each loop-free

path is executed or not. Different test cases, varying only in the date inputs, generate different *path spectra*. The path spectrum from a test case is a bar graph with one bar for each loop-free path in the program, of length zero (if the path was not covered) or one (if it was covered.) They propose comparing the path spectra of pre-2000 tests with those of post-2000 tests to identify year 2000 problems.

The main differences between their approach and the one in this paper are in the use of loop-free path spectra instead of block execution counts, and in the display and analysis methods proposed. Loop-free paths may identify more year 2000 faults than basic blocks since the granularity of analysis is lower. Reps *et al.* (1997) give an example showing how this may be true. However, for a technique to be incorporated into a practical tool, it is also important that the results should be easy to understand. Loop-free paths are a less intuitive concept than basic blocks, so it may be more difficult for practising programmers to grasp quickly the meaning of a difference between two path spectra. It is relatively easy to understand statements such as 'block X is executed four times in pre-2000 data and only two times with similar post-2000 data'. From our experience, we believe that the analysis and display methods presented in the following sections could be quickly exploited by most practising programmers.

3. THE EXAMPLE PROGRAMS

Since the year 2000 problem was one of the main motivations for this research, we illustrate each method by applying it to five C programs that encode dates in different ways. One of the reasons that the year 2000 problem is difficult is that programmers have invented many different ways of encoding date information, not all of which are easy to identify. We chose encodings from our experience, that have been mentioned in the literature, or that appeared in responses to our posting on the 'comp.software.year-2000' newsgroup.

The five C programs all perform the same task: computation of the compound amount of a loan. Each program passes four command line arguments:

- amount of loan (e.g., USA$500.00)
- loan interest rate, compounded daily (e.g., 0.001)
- start date of the loan (e.g., 15/01/85)
- end date of the loan (e.g., 15/06/91)

Each program outputs the number of days in the loan period and the amount of the loan including interest.

The five encodings are as given in Table 1. All but the `time.c` program have the year 2000 'bug', in that they read two digit years and thus perform incorrect computations when the input is '00'. The `time.c` program is an example of a common one-line fix for the year 2000 problem using the 100-year window approach. Any year input that is less than 76 is incremented by 100 so an input such as 12/05/02 would be interpreted as 12 May 2002.

4. IDENTIFYING INPUT SUBDOMAINS

The concept of 'input domains' is a common one in testing theory (Beizer, 1990, chapter 6). Testers examine the specification to partition the input space into subdomains within which the

Table 1. The five versions of the compound amount program

Program	Description
`camt.c`	Uses a 'Julian day' approach in which dates are stored as the integer number of days since 1 January of a given base year. The algorithm for computing the loan period loops through each year since the base, checking for leap years, and through each month adding the different month lengths.
`yy365.c`	Uses a one-year cycle to store dates as a single integer, e.g., 01/01/95 is stored as 95001; 08/03/95 is stored as 95067 (the 67th day of 1995). Again, loops are used to count leap years and month lengths.
`cycle.c`	Uses a four-year cycle of 1461 days beginning on the most recent leap year (relative the date to be encoded). Stores dates as a single integer, e.g., 27/03/96 is stored as 960087; 28/09/97 is stored as 960637 (the 637th day of the four year cycle beginning on the first day of the most recent leap year—01/01/96).
`time.c`	Uses the date functions from the standard C library declared in the header `time.h`, to calculate the number of days between two dates. Parts of this code were taken (with permission) from a program written by Karlon West at Nortel.
`comp_method.c`	Uses an alternative 'Julian day' approach that stores each date as the number of days since 1 January, 1900. Instead of loops to check leap years and month lengths, it computes the number of leap years directly, and does a look-up in a table of month lengths.

program is expected to behave in the same way. Test cases are chosen inside and at the boundaries of the subdomains.

For the year 2000 problem, instead of a specification we start with a legacy program and try to identify the date subdomains. Does the program execute the same way regardless of the date input? Or are there different subdomains, with different processing in certain months, in leap years or in the year 2000 itself? We can get this information from the execution count vectors.

To analyse the program for a specific date input x_i we fix values for the remaining inputs and run a series of tests with different values of x_i. As previously described, the test tool gives us an execution count vector for each test. All the dates x_i that generate identical execution count vectors are grouped into one subdomain since processing is 'the same'.

This kind of analysis can give a maintainer very interesting insight into the program. First, he or she can determine if a particular input data item has any complex interactions with the code. The simplest case is that there is only one subdomain so that x_i has no effect on code execution. More complex data items may produce a small number of subdomains, say two to four, which indicates to the maintainer that the program does some simple case analysis involving these data.

The five date encoding programs described earlier were used to illustrate this method for identifying subdomains. The loan start date was fixed at 01/01/95 and the end date was increased by one day every test case, beginning at 01/01/95 and ending at 31/12/00, for 2192 test cases spanning six years. The results are summarized in Table 2.

The `time.c` version of the program using the C library function `time.h` shows just two subdomains, one corresponding to the years up to 2000 and one for years 2000 and beyond. As previously mentioned, this version has a fix for the year 2000 problem that adds 100 to input years

Table 2. Subdomains in the five program versions

Program version	Number of subdomains
time.c	2
comp_method.c	2
cycle.c	48
yy365.c	70
camt.c	70

```
dy = (365 * y + ((y-1)/4));
if (0 == y % 4)
  dm = Leap[m];    <= Block 2
else
  dm = nonLeap[m]; <= Block 4
dd = d;
newDate = dy + dm + dd;
```

Figure 2. Leap year algorithm in `comp_method.c`

that are less than 1976. Other than that fix, there is no processing that depends on the input date because all date calculations are done by the `mktime()` library function and thus are not part of the code of the program itself.

The `comp_method.c` version shows two subdomains, which turn out to correspond to normal years and leap years. The algorithm used to determine how many days to d/m/y since 1 January of the year 1900 is shown in Figure 2.

In leap years the line labelled ‘Block 2’ is executed while in normal years ‘Block 4’ is taken. This will give two different count vectors and thus two subdomains. (The convention used in this paper is to identify basic blocks by the reference numbers assigned by the *ATAC* tool. These reference numbers are, roughly, consecutive within each C function.)

The `cycle.c` version resulted in 48 count vectors, one for each month of the four year leap-year cycle. This program version contains the code shown in Figure 3 to determine where we are in the four years of each cycle.

It also includes the days-in-month loop of Figure 4 to help compute the number of days since the beginning of the year.

The loop of Figure 4 will give different execution count vectors depending on the value of month, while the if statements of Figure 3 will execute differently in the four years of the leap-year cycle. In combination they give a total of 48 different count vectors and thus 48 subdomains.

Note that the number of subdomains can grow quite rapidly if the data item controls a loop, since we will have a different count vector for every possible number of loop executions. Also if, as in this case, the data item affects control in two separate ways, then the number of subdomains may be multiplied.

The number of subdomains may be an interesting kind of complexity metric. A single subdomain


```
/* the year of the date to be encoded is a leap year */
if (0 == y % 4)
  {
   numDays = daysSince(m, d, y);
   cycle = y;
  }

/* the year of the date to be encoded is one year after
 * a leap year */
if (0 == (y - 1)  % 4)
  {
   numDays = daysSince(m, d, y);
   numDays = numDays + 366;
   cycle = y - 1;
  }

/* the year of the date to be encoded is two years after
 * a leap year */
if (0 == (y - 2)  % 4)
  {
   numDays = daysSince(m, d, y);
   numDays = numDays + 366 + 365;
   cycle = y - 2;
  }

/* the year of the date to be encoded is three years after
 * a leap year */
if (0 == (y - 3)  % 4)
  {
   numDays = daysSince(m, d, y);
   numDays = numDays + 366 + 365 + 365;
   cycle = y - 3;
  }
```

Figure 3. Leap year algorithm in `cycle.c`

indicates no interaction between the data and control flow, while a small number hints at relatively few cases to consider. A large number of subdomains may indicate either an important data item that has widespread effects in many parts of the program, or else a loop control variable whose value directly affects the number of times a loop is executed.

While it is always rash to measure program quality from a single metric, it is probably true that a maintainer would much prefer to deal with the `time.c` version. There, library code takes care of most date manipulation. A top second choice is the `comp_method.c` version where table look-up replaces a complex algorithm for counting days since 1 January.


```
for (m=1;m<month;m++) {
    switch(m) {
       case 9:  /* September */
       case 4:  /* April     */
       case 6:  /* June      */
       case 11: /* November */
                days = days + 30;
                break;
       case 2: /* February   */
                { if (0 == year % 4)
                    days = days + 29;
                  else
                    days = days + 28;
                  break;
                 }
      default: /* all the rest ... */
                { days = days + 31;
                  break;
                }
    } /* end switch */
```

Figure 4. Days of the month algorithm in `cycle.c`

5. PATTERNS OF BLOCK EXECUTION

If the subdomain analysis indicates that the code is data sensitive, then the obvious next step in program comprehension is to look at the blocks whose execution count varies. A simple plot can quickly provide insight into the program.

A tool to aid in data sensitivity analysis could first present the programmer with a list of blocks that exhibit any sensitivity. A mouse click on a block in the list could either show the code, or else a plot of the execution counts against the date. We have been using the *XGobi* tool, originally developed at Bellcore, to produce such screen plots (Buja, Cook and Swayne, 1996). For off-line analysis similar plots can be generated by spreadsheet software such as Microsoft *Excel*. (Graphs based on the *Excel* plots have been used for the figures in this paper.)

As a first example Figure 5 graphs the execution counts of the two blocks highlighted in Figure 2. The first step for Block 2 comes exactly when the input date reaches 1 January of the leap year 1996, and lasts one year. The next step comes when the date arrives at the year 2000, again a leap year. It is almost unnecessary to look at the code in Figure 2 to guess that this is a leap-year algorithm. The pattern of Block 4 is just the reverse of Block 2 so it is clear that it is handling the non-leap-year case.

As a second example, Figure 7 graphs the execution count of Block 8 in Figure 6, which counts days in 'non-leap years' that are totally contained in the loan period. The block is in a `for` loop with `x` initialized to `y1`, the start year of the loan, which is 1995 in our tests. The loop executes while `x` is less than `y2`, the end year of the loan. As the number of years in the loan period grows,

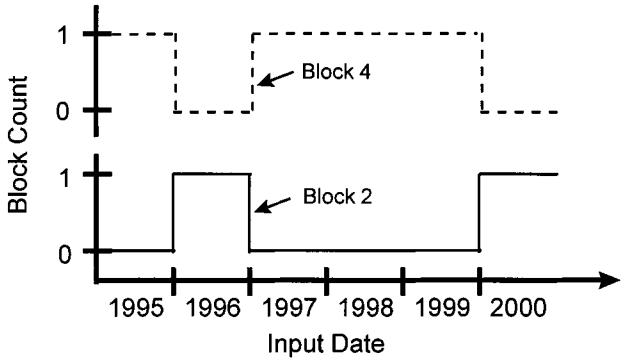

Figure 5. Execution counts for a simple leap-year algorithm

```
for (x=y1; x < y2; x++)
 {
  if (0 == x % 4)
    diff = diff + 366;
  else
    diff = diff + 365; <= Block 8
 }  /* end for */
```

Figure 6. Accumulating days over several years

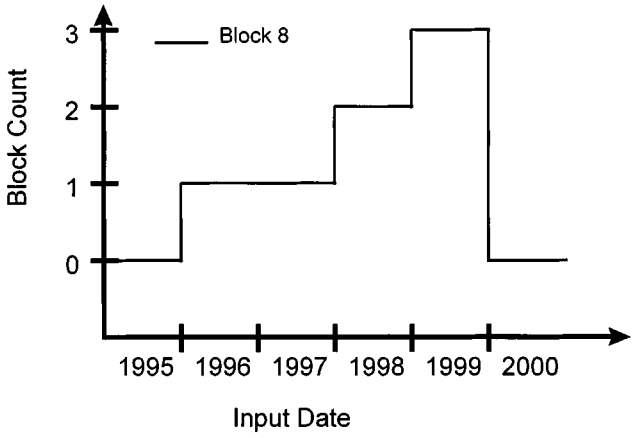

Figure 7. A leap-year algorithm with a bug!

so does the number of 'non-leap' years. For example, when `y1` and `y2` are both 1995, Block 8 is not executed because the loop body is not entered. Thus, the block count in Figure 7 is zero for 1995. In the following year, when `y2` is 1996, the loop is executed once with `x` = 1995 which gives the block count of one shown in Figure 7. Then when `y2` becomes 1997, the loop is executed twice

Table 3. Amplitude spectrum for blocks 2 and 4 from Figure 2

Frequency (1/days)	Period (days)	Amplitude spectrum	
		Block 2	Block 4
0.0000	Constant	257.000	1 791.000
0.0007	1 461	231.187	231.187
0.0014	731	162.973	162.973
0.0021	487	76.118	76.118
0.0027	365	1.000	1.000
0.0034	292	46.800	46.800
0.0041	244	54.319	54.319
0.0048	209	32.214	32.214
0.0055	183	1.000	1.000
0.0062	162	26.310	26.310
0.0068	146	32.585	32.585
0.0075	133	20.238	20.238
0.0082	122	1.000	1.000
0.0089	112	18.427	18.427
0.0096	104	23.268	23.268
0.0103	97	14.647	14.647
0.0110	91	1.000	1.000

(once with `x` = 1995 and once with `x` = 1996), but Block 8 is only executed for the non-leap year `x` = 1995. This is why the block count in Figure 7 is one for both 1996 and 1997. A similar analysis applies for 1998 and 1999. However, in the year 2000 the count falls to zero! The graph reveals a year 2000 fault; since `y2` is 00, the loop body in Figure 6 is not being executed.

To sum up, a programmer analysing unfamiliar code for year 2000 problems could first run an automatically generated series of tests, using a test coverage tool to collect execution counts. The counts would be compared to detect date sensitive code, and then displayed to give insight into algorithms and possible bugs. The detection method would let the programmer focus quickly on dangerous code, and the display would then take advantage of the human capability to recognize and analyse graphic patterns.

6. FOURIER ANALYSIS

If a maintainer is dealing with a large volume of date sensitive code, then Fourier analysis of the execution count data might be considered as an additional way of identifying patterns. Fourier analysis is a well-known technique for studying signals and is described in many standard texts (for example, Houts and Alkin (1991, chapter 2)). The amplitude spectrum of a signal highlights the frequencies that contain most of the signal's power.

As an example, Table 3 shows the first few entries in the amplitude spectrum for four years of counts from two basic blocks. These are the blocks of the leap-year computation in Figure 2 whose block counts were graphed in Figure 5.

Note that, except for the constant entry, the amplitude spectra of the two blocks are identical, although the graphs in Figure 5 are inverses. It is well known that phase shifts do not affect the

Fourier transform, while multiplication by a constant only affects the constant term. Thus, different basic blocks doing similar things may have similar amplitude spectra. The spectrum shown in Table 3 turns out to cover many cases of 'something happening one year out of four', and it occurred for several basic blocks involved in leap-year computations in different versions of the loan calculator. It would be possible to create a library of such spectra and match them automatically with the execution count data from a large system to provide quick clues to a year 2000 programmer.

7. EXPLOITING CHARACTERISTICS OF THE DATA

The methods presented previously made no use of any knowledge about the data themselves. But many kinds of data have known regularities: angles are 'the same' when you add 360 degrees, customer account codes have only certain defined values and programs often do something special at powers of two.

For dates, it has been pointed out that there is an interesting and potentially useful cycle of 28 years. If a date is shifted by exactly 28 years, the day of the week and the position in the leap-year cycle are exactly the same. (Of course this regularity is broken in years such as 1900 and 2100 which are not leap years, although divisible by four with a zero remainder. However, the year 2000 is an exception to the exception and is a leap year.)

One method for analysing code for year 2000 problems is to take two test cases that differ only in that the second has all the dates shifted back 28 years. The output from the two test cases should be identical (Royer, 1997). We suggest that a useful extension to this method may be to compare the execution count vectors from the two tests, as well as comparing the output. Thus, we can expose any internal differences in execution that might signal a year 2000 fault, even if the difference is masked in the output. (It is not uncommon for users to discover a fault that has been executed in many test runs, but the fault was not observable in the output from those particular tests.)

The objective is to exploit the 28-year regularity to identify the safe and unsafe domains for the program. We run two series of tests, one with dates in a presumed safe range and the other with dates 28 years later, incorporating the year 2000.

To illustrate this method we used the same five programs described earlier. In series 1 the loan start date ranged from 01/01/71 to 31/12/73 with the loan end date 100 days later. Series 2 used loan start dates 28 years later, from 01/01/99 to 31/12/01 again with the loan end date 100 days after the start. Each series has 1096 test cases. The loan period included 01/01/00 in cases numbered 265 to 364, and these are the tests that are expected to show anomalies since the length of the loan will be calculated incorrectly.

The method worked as anticipated for the versions `camt.c` and `yy365.c`. Both programs gave the correct result for series 1, but gave different (wrong) answers for tests 265 to 364, with a corresponding difference in the execution count vectors. In each case, a quick examination of the blocks that were executed differently pointed to the variables that contained dates. We believe a maintainer would have had little difficulty in using this information to design a fix.

For the versions `cycle.c` and `comp_method.c`, the results were a bit more ambiguous. Both programs gave the correct result for Series 1 and wrong answers for tests 265 to 364 of series 2. Additionally, `comp_method.c` computed a loan period of only 99 days for tests 631 to 730, in which the loan start date is in year 00, while the end date is in year 01. The cause was a hitherto unknown fault in the code that counted leap years.

However, the execution count vectors showed no differences between the two series. The algorithm executes in exactly the same way in the safe and unsafe periods—it just gets the wrong answer.

Finally, the `time.c` version gave the wrong answer for the tests of series 1! The version of the library `mktime()` function we used is correct for dates only between 1970 and 2036. The 'fix' to the year 2000 problem shifts dates less than 1976 by adding 100, so that the series 1 tests with dates in 1971 and 1972 were shifted to 2071 and 2072. The library function returned an error code which was not caught, and the program produced the wrong answer. So perhaps this example shows a peril of using the 28-year shift as a test oracle. In this case it flags the program as faulty, when really it is correct for the years after 1976 that are of interest.

The 28-year shift method provides an additional check on execution counts that may, in some cases, detect year 2000 problems with relatively few test cases. However, it cannot be relied upon to always flag incorrect code, since the execution may be the same even if the wrong result is being computed. Furthermore, as in the `time.c` version, it may occasionally flag code which is actually correct.

8. SCALE-UP ISSUES

This paper has presented several techniques that make use of traces from a test coverage monitor to locate and analyse year 2000 problems in existing code. Since all the examples given are very small, it is appropriate to discuss the difficulties that would be encountered in applying these techniques to a real industrial software system.

The first requirement is for a good test coverage monitor, capable of handling the entire system or, at the least, major subsystems. This requirement is becoming easier to satisfy since efficient tools are now becoming available that can handle systems up to several hundred thousand lines of code. For example, the *ATAC* tool of this study has been successfully used in Bellcore on several applications which are even larger than this size. The usual experience seems to be that, for large systems, test coverage tools need to be adapted slightly to each system, but that such adaptation is possible except in environments with very tight memory or performance constraints.

The second requirement is a good test harness that can automatically run large numbers of similar tests. This requirement may be harder to satisfy for much legacy code. Unfortunately, test harnesses need to be developed specifically for each system, and many legacy systems were created by organizations that did not devote resources to facilitating lifetime testing. The development of such a harness after the fact can require considerable effort. Perhaps the availability of the analysis methods described here will help to motivate future developers to give more emphasis to test environments.

Finally, we should consider if the analysis techniques presented here will scale up to systems with hundreds of thousands of basic blocks. Fortunately, the performance of all our analysis algorithms should be linear in the number of basic blocks handled. The computation of input subdomains as described in Section 4 can be accomplished by simply sorting the execution count vectors, so that all the tests with identical vectors will wind up together. Recall that there is one execution count vector for each test case, and that the vector has one integer for each basic block in the program. In the 'big-O' notation used in the analysis of algorithms, the time T for such a sort is typically

$$T = O(CN \log(N)) \tag{1}$$

where N is the number of vectors and C is the time to compare two of them (Lewis and Denenberg, 1991, pp. 393–396). In this case, N is the number of test cases and is fixed independent of the program being analysed. As previously mentioned we always used 2 192 cases, one for each day in a six-year range. The factor which does vary as the program gets larger is C, the vector comparison time. However, vector comparison is just linear in vector size, so overall performance will be linear in the number of basic blocks in the program.

Similarly, the pattern display method of Section 5 and the Fourier analysis method of Section 6 are both linear in the number of basic blocks. For both methods we first simply scan through the blocks to find which ones show date sensitivity. Then a fixed-size plot or Fourier analysis is performed for each such block. Finally, the 28-year shift method similarly involves a comparison of pairs of execution count vectors. Each comparison would take linear time in the length of the vector.

9. CONCLUSIONS

This paper has presented several dynamic methods for analysing code for year 2000 faults. However, we should point out that the methods described in this paper are not limited to dates or to the year 2000 problem. Other important program plans may be located by studying the variations of block execution counts as input data changes. For example, in Agrawal *et al.* (1998) a case study is given that shows how special optimization code in a Bellcore subroutine library was located by running tests with data structures of different sizes.

So while the year 2000 problem was the original motivation, this paper is really about analysing *data sensitivities* of code, not just date sensitivities. The paper is obviously only a starting point. The examples are all small programs and not necessarily typical of phenomena found in real systems. However, we think they give us reason to believe that program comprehension techniques based on examining execution counts may be of both theoretical and practical benefit.

From the standpoint of theory, the methods described here give a different perspective on a program. We can ask questions such as:

- to which inputs is this program sensitive?
- how does the program partition the input domain into subdomains?
- is the sensitive code localized in one area, or is it spread through many different modules?

For the practical maintainer, dynamic analysis can provide a range of techniques for tracking down problems and understanding code. The analysis points the maintainer directly to the code that is sensitive to the data item. In a large system, such as the insurance example given in the introduction, this is probably the most important benefit, since other methods of searching for such code may be either very costly or dependent on naming and commenting conventions which are not always followed.

Graphical display and Fourier analysis are techniques that may help to detect common programming cliches, and even some bugs, quite quickly. The display technique seems particularly promising, since it combines the computer's number crunching abilities with the human capability for recognizing patterns in images.

Techniques such as the 28-year date shift can be useful for some programs. The shifted test cases provide a test oracle that gives 'correct' output for comparison purposes. As we have seen, in some

cases, but not all, the execution count comparisons can also give good starting points for code exploration.

A clear advantage of these dynamic analysis methods is the ease with which tools can be constructed using such existing building blocks such as *ATAC*, *XGobi* and *Excel*. The main disadvantage of the methods is the need to run a large number of test cases. The cases follow a simple pattern, so it is easy to generate the required test data automatically. However, legacy systems often do not have good test harnesses that facilitate running many tests and checking the output. If not already available, development of such a harness requires a non-trivial investment.

Acknowledgements

We would like to thank Melanie Sutton of the Computer Science Department of the University of West Florida for her suggestion of the graphical analysis method presented in Section 5, as well as for locating the *XGobi* display tool used there. We would also like to thank Karlon West at Nortel for providing the code on which the `time.c` example was based.

This work was supported by the Software Engineering Research Center, an NSF-supported industry–university collaborative research centre, under NSF grant EEC–9418762 and others. Current industrial affiliates of the centre are: Nortel; Dynamix, Inc.; Motorola, Inc.; British Telecom; Bell Communications Research; Northrop Grumman ESID; Northrop Grumman ESD; and Army Research Lab.

References

Agrawal, H., Alberi, J. L., Horgan, J. R., Li, J. J., London, S., Wong, W. E., Gosh, S. and Wilde, N. (1988) 'Mining system tests to aid software maintenance', *IEEE Computer*, **31**(7), 64–73.

Beizer, B. (1990) *Software Testing Techniques*, International Thompson Computer Press, Boston MA, 550 pp.

Bohner, S. A. (1996) 'Impact analysis in the software change process: a year 2000 perspective', in *Proceedings International Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos CA, pp. 42–51.

Buja, A., Cook, D. and Swayne, D. F. (1996) 'Interactive high-dimensional data visualization', *Journal of Computational and Graphical Statistics*, **5**(1), 78–99. The *XGobi* tool is available at URL: http://lib.stat.cmu.edu/general/XGobi

Gowan, J., Jesse, C. and Mathieu, R. (1999) 'Y2K compliance and the distributed enterprise', *Communications of the ACM*, **42**(2), 68–73.

Hart, J. M. and Pizzarello, A. (1996) 'A scaleable, automated process for year 2000 system correction', in *Proceedings 18th International Conference on Software Engineering, ICSE'96*, IEEE Computer Society Press, Los Alamitos CA, pp. 475–484.

Holmes, L. C. (1997) 'Data aging for testing the year 2000 solution', *CrossTALK – The Journal of Defense Software Engineering*, **10**(10), 16–17.

Horgan, J. R. and London, S. A. (1992) 'ATAC: a data flow coverage testing tool for C', in *Proceedings Symposium on Assessment of Quality Software Development Tools*, IEEE Computer Society Press, Los Alamitos CA, pp. 2–10.

Horgan, J. R., London, S. and Lyu, M. R. (1994) 'Achieving software quality with testing coverage measures', *IEEE Computer*, **27**(9), 60–69.

Houts, R. and Alkin, O. (1991) *Signal Analysis in Linear Systems*, Saunders College Publishing, Philadelphia PA, 413 pp.

Kappelman, L., Fent, D., Keeling, K. and Prybutok, V. (1998) 'Calculating the cost of year-2000 compliance', *Communications of the ACM*, **41**(2), 30–39.

Lefkon, D. (1997) 'Seven work plans for year 2000 upgrade project', *Communications of the ACM*, **40**(5), 111–113.

Lewis, H. and Denenberg, R. (1991) *Data Structures and their Algorithms*, Harper Collins, New York NY, 509 pp.

Marcoccia, L. (1998) 'Building infrastructure for fixing the year 2000 bug: a case study', *Journal of Software Maintenance*, **10**(5), 333–352.
Martin, R. A. (1997) 'Dealing with dates: solutions for the year 2000', *IEEE Computer*, **30**(3), 44–51.
Newcomb, P. H. and Scott, M. (1997) 'Requirements for advanced year 2000 maintenance tools', *IEEE Computer*, **30**(3), 52–57.
Reps, T., Ball, T., Das, M. and Larus, J. (1997) 'The use of program profiling for software maintenance with applications to the year 2000 problem', in *Proceedings ESEC/FSE 97*, published as Vol 1301 of *Lecture Notes in Computer Science*, Springler-Verlag, Berlin, pp. 432–449.
Royer, T. C. (1997) 'Evaluating systems for year 2000 Compliance', available at URL: http://www.mitre.org/research/y2k/docs/TestY2K.pdf
Rubin, H. (1999) 'Bracing for the millennium', *IEEE Computer*, **31**(1), 51–56.
Sharon, D. (1997) 'Year 2000 tool classification scheme', *IEEE Software*, **14**(4), 107–111.
Thomas, T. (1998) 'Myths of the year 2000', *The Computer Journal*, **41**(1), 67–70.
Wilde, N. and Scully, M. (1995) 'Software reconnaissance: mapping program features to code', *Journal of Software Maintenance*, **7**(1), 49–62.
Zvegintzov, N. (1997) 'A resource guide to year 2000 tools', *IEEE Computer*, **30**(3), 58–63.

Authors' biographies:

Norman Wilde is a Professor of Computer Science at the University of West Florida in Pensacola in Florida. Norman spent a number of years working overseas in developing countries in academic positions, with the World Health Organization and as an independent systems consultant. Since 1986 he has been working with the Software Engineering Research Center on a series of projects in software maintenance and particularly in the area of dependency analysis and software reconnaissance. He received his Ph.D. in Mathematics and Operations Research from the Massachusetts Institute of Technology in 1971. His email address is: nwilde@uwf.edu

Randy L. Justice is a Senior Oracle Data Base Administrator for DynTel Corp. at Saufley Field Pensacola, and an adjunct instructor at Pensacola Junior College. Prior to that, he worked with Technical Software Services in Pensacola as a Systems Analyst. Randy received a M.S. in Software Engineering from the University of West Florida in May 1998 and a B.S. in Computer Science in 1991. His email address is: Randy.Justice@smtp.cnet.navy.mil

Kristin Blackwell is working as a Software Engineer for Motorola, Inc. in Arlington Heights, Illinois in the Cellular Infrastructure Group. She graduated from the University of West Florida in 1997 with a Masters degree in Computer Science. At the University, Kristin worked on a graduate project to locate and understand code whose execution depends on date inputs. Her email address is: blackwll@cig.mot.com

W. Eric Wong is a Research Scientist in the Software Environment Research Group at Bellcore. Eric's research interests include program analysis techniques to help software development, especially software testing, debugging and understanding. He is also interested in software architecture and design metrics analysis. In 1997, he received the Quality Assurance Special Achievement Award from Johnson Space Center, NASA. Eric received his B.S. in Computer Science from Eastern Michigan University, and his M.S. and Ph.D. in Computer Science from Purdue University. His email address is: ewong@bellcore.com

